

Portable Performance in Real Applications Using Generated Code

DOE Centers of Excellence Meeting

David Richards,
Pei-Hung Lin, Prashant Rawat, Louis Noel Pouchet,
Saday Sadayappan, Dan Quinlan

April 21, 2016

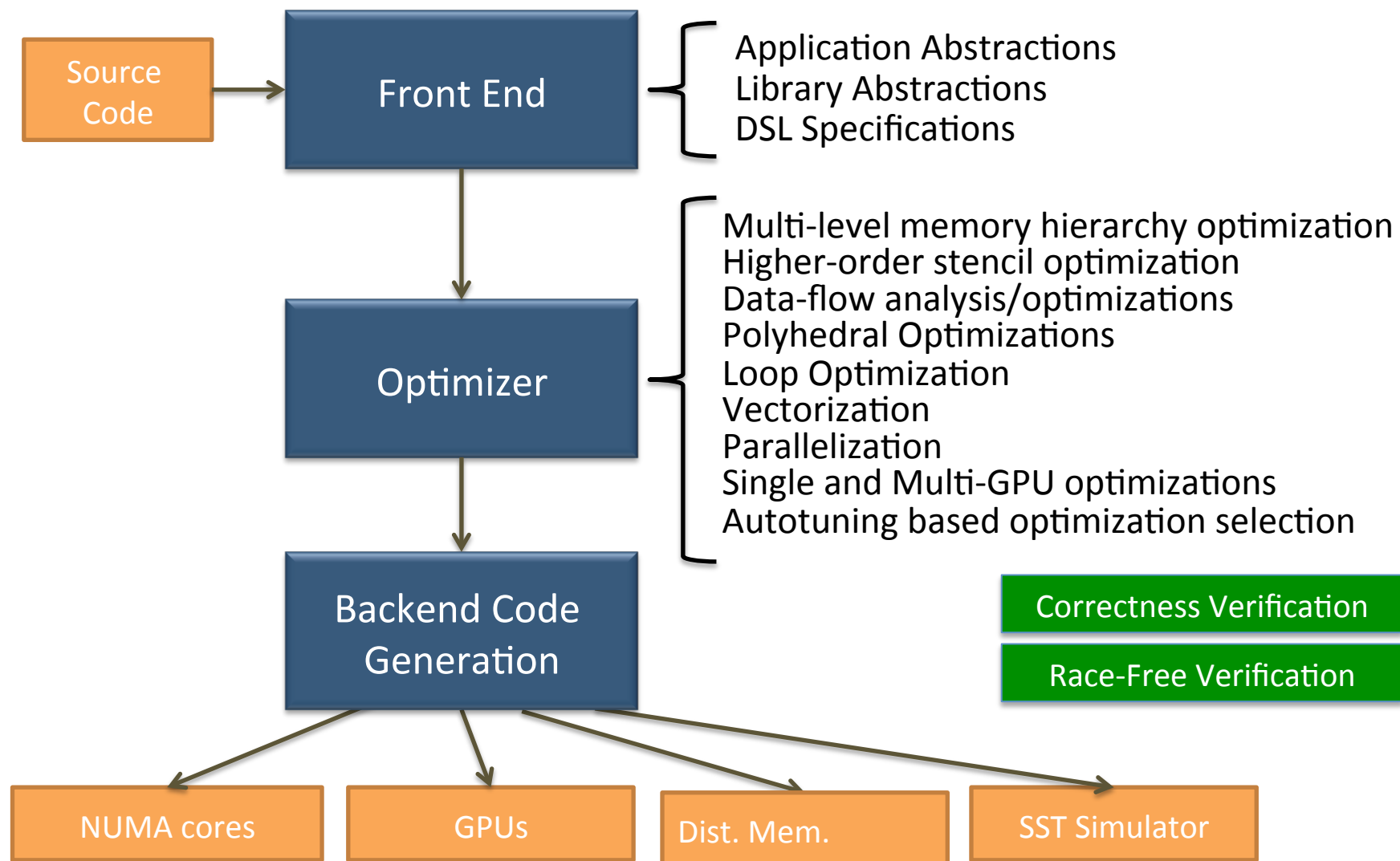


LLNL-PRES-690278

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

 **Lawrence Livermore
National Laboratory**

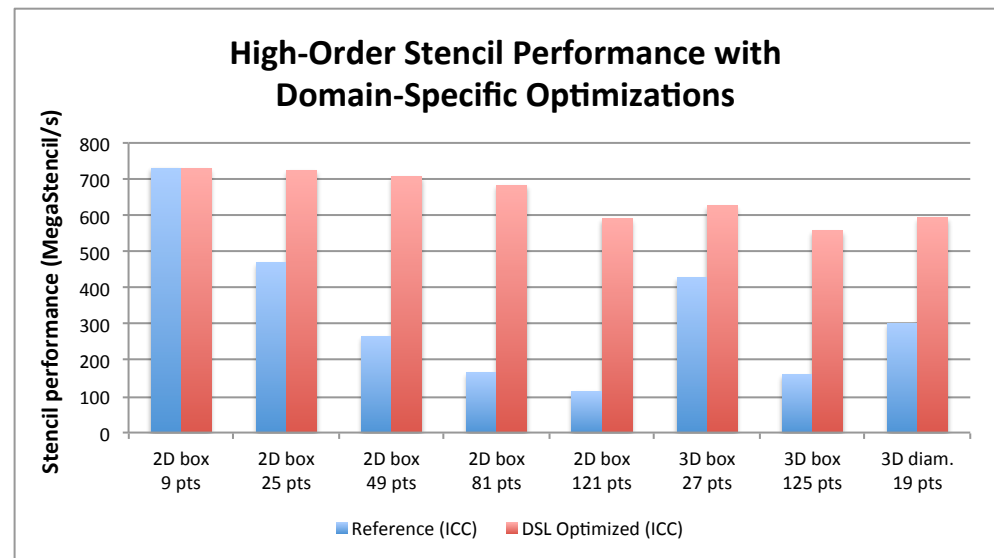
Outline of the D-TEC Flow



5. Demonstrate how to extend for non-trivial app. evolution

Higher Order Stencils in ROSE/PolyOpt

- **Significance / Impact**
 - High-order stencils arise in high-accuracy numerical solution approaches for PDEs
 - Chombo and Overture applications make use of high-order stencils, but current implementations pay a high compute cost for increased stencil order
 - A new domain-specific optimization has enabled significantly enhanced performance for high-order stencils, on multi-core processors
 - Implication: more accurate solution schemes using high-order stencils can be run in about the same time as one with lower order stencils and lower accuracy
- Generates high-performance codes of **> 10,000 lines** automatically from **< 20 lines** DSL description



U.S. DEPARTMENT OF
ENERGY

LLNL-PRES-690278

Office of
Science

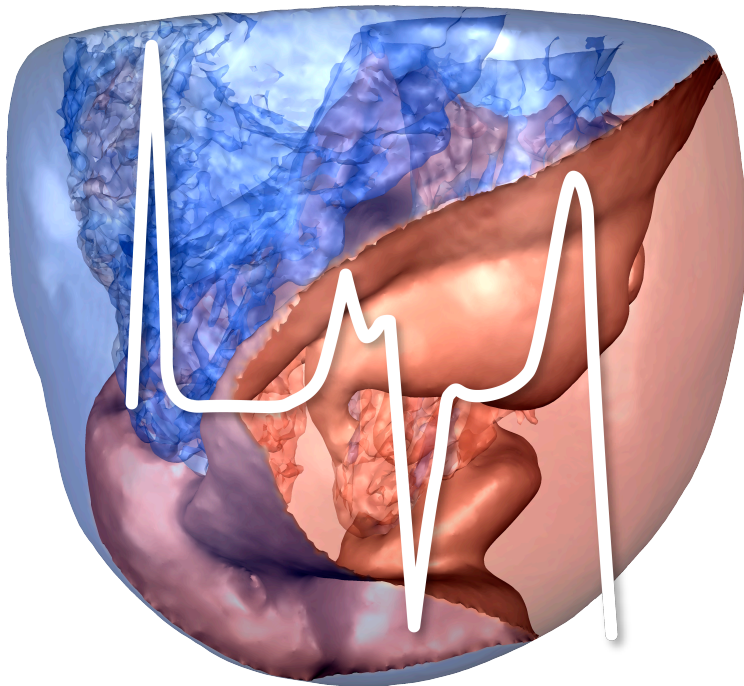


Lawrence Livermore
National Laboratory

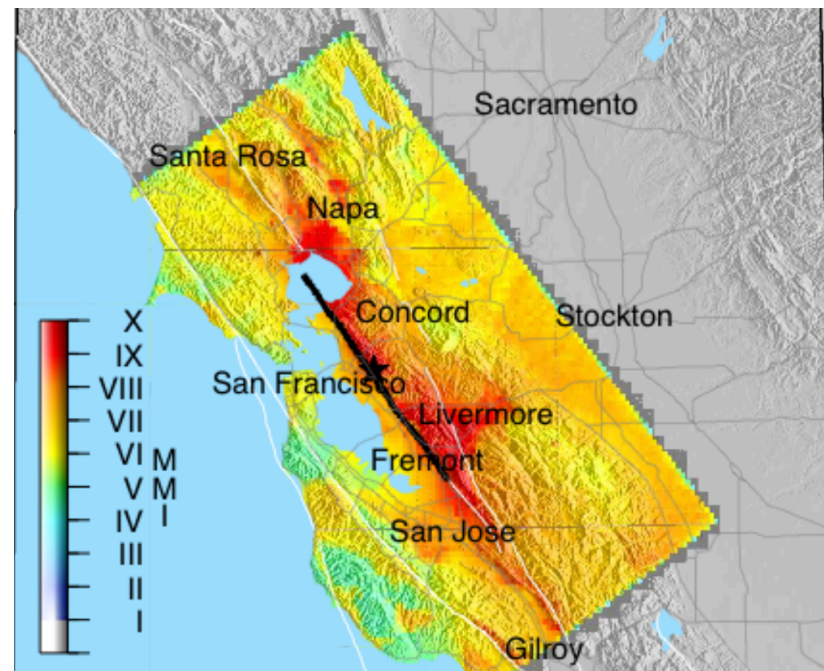


D-TEC

Cardioid and SW4



Cardiac Electrophysiology
1st Order 19 point stencil



Seismic Wave Propagation
4th order 125 point stencil

Both codes use structured grids and handle anisotropic inhomogeneous materials.

Generated GPU Code for Cardioid

- **A variable coefficient stencil in the diffusion code**
 - Order-1 bandwidth-bound stencil on a 102 x 37 x 17 domain
 - Original code with OMP (4 threads) achieves 3.76 Gflop/s
- **Efficient GPU code can be generated by a code generator**
 - With sufficiently high occupancy, and no register spills
 - A naïve generated code achieves 34.50 Gflop/s
- **Roofline Model Analysis:**
 - Operational intensity (ops/byte) : 0.18
 - Maximum achievable Gflop/s: peak bandwidth * 0.18 = 51.91 Gflop/s
- **Performance Analysis:**
 - Generated code achieves 66.63% of the “roofline” maximum
 - Achieved global memory load efficiency: 76.2%
 - Achieved global memory store efficiency: 86.21

SW4 Kernel

- Simplest kernel from rhs4th3fort routine in SW4
- Complex high-order stencil
- Five 3D input arrays, three 3D output arrays
- Very high arithmetic intensity:
over 650 floating-point operations => over 10 FLOPs/byte
- Very different from Jacobi stencils frequently evaluated for code generation

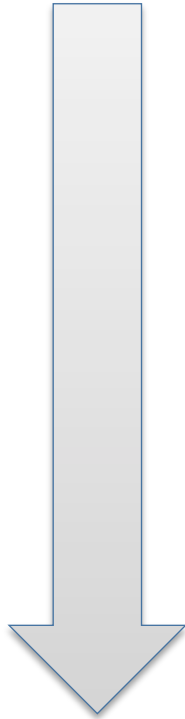
SW4 stencil code

```
for( k= k1; k <= k2 ; k++ )
{
    for( j=jfirst+2; j <= jlast-2 ; j++ )
    {
        for( i=ifirst+2; i <= ilast-2 ; i++ )
        {

            /* from inner_loop_4a, 28x3 = 84 ops */
            mux1 = mu(i-1,j,k)*strx(i-1)-
                tf*(mu(i,j,k)*strx(i)+mu(i-2,j,k)*strx(i-2));
            mux2 = mu(i-2,j,k)*strx(i-2)+mu(i+1,j,k)*strx(i+1)+
                3*(mu(i,j,k)*strx(i)+mu(i-1,j,k)*strx(i-1));
            mux3 = mu(i-1,j,k)*strx(i-1)+mu(i+2,j,k)*strx(i+2)+
                3*(mu(i+1,j,k)*strx(i+1)+mu(i,j,k)*strx(i));
            mux4 = mu(i+1,j,k)*strx(i+1)-
                tf*(mu(i,j,k)*strx(i)+mu(i+2,j,k)*strx(i+2));

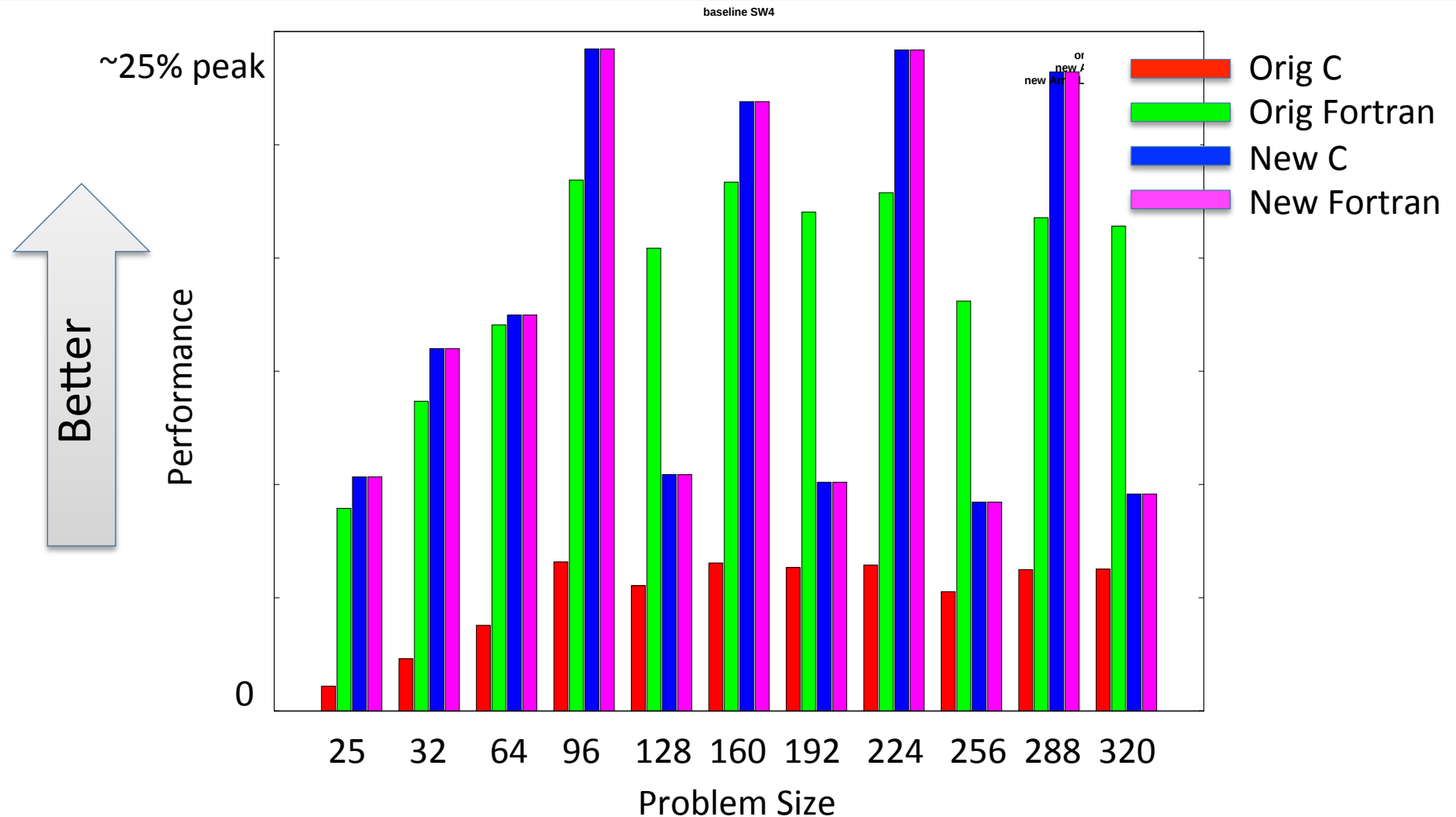
            muy1 = mu(i,j-1,k)*stry(j-1)-
                tf*(mu(i,j,k)*stry(j)+mu(i,j-2,k)*stry(j-2));
            muy2 = mu(i,j-2,k)*stry(j-2)+mu(i,j+1,k)*stry(j+1)+
                3*(mu(i,j,k)*stry(j)+mu(i,j-1,k)*stry(j-1));
            muy3 = mu(i,j-1,k)*stry(j-1)+mu(i,j+2,k)*stry(j+2)+
                3*(mu(i,j+1,k)*stry(j+1)+mu(i,j,k)*stry(j));
            muy4 = mu(i,j+1,k)*stry(j+1)-
                tf*(mu(i,j,k)*stry(j)+mu(i,j+2,k)*stry(j+2));

            muz1 = mu(i,j,k-1)*strz(k-1)-
                tf*(mu(i,j,k)*strz(k)+mu(i,j,k-2)*strz(k-2));
            muz2 = mu(i,j,k-2)*strz(k-2)+mu(i,j,k+1)*strz(k+1)+
                3*(mu(i,j,k)*strz(k)+mu(i,j,k-1)*strz(k-1));
```

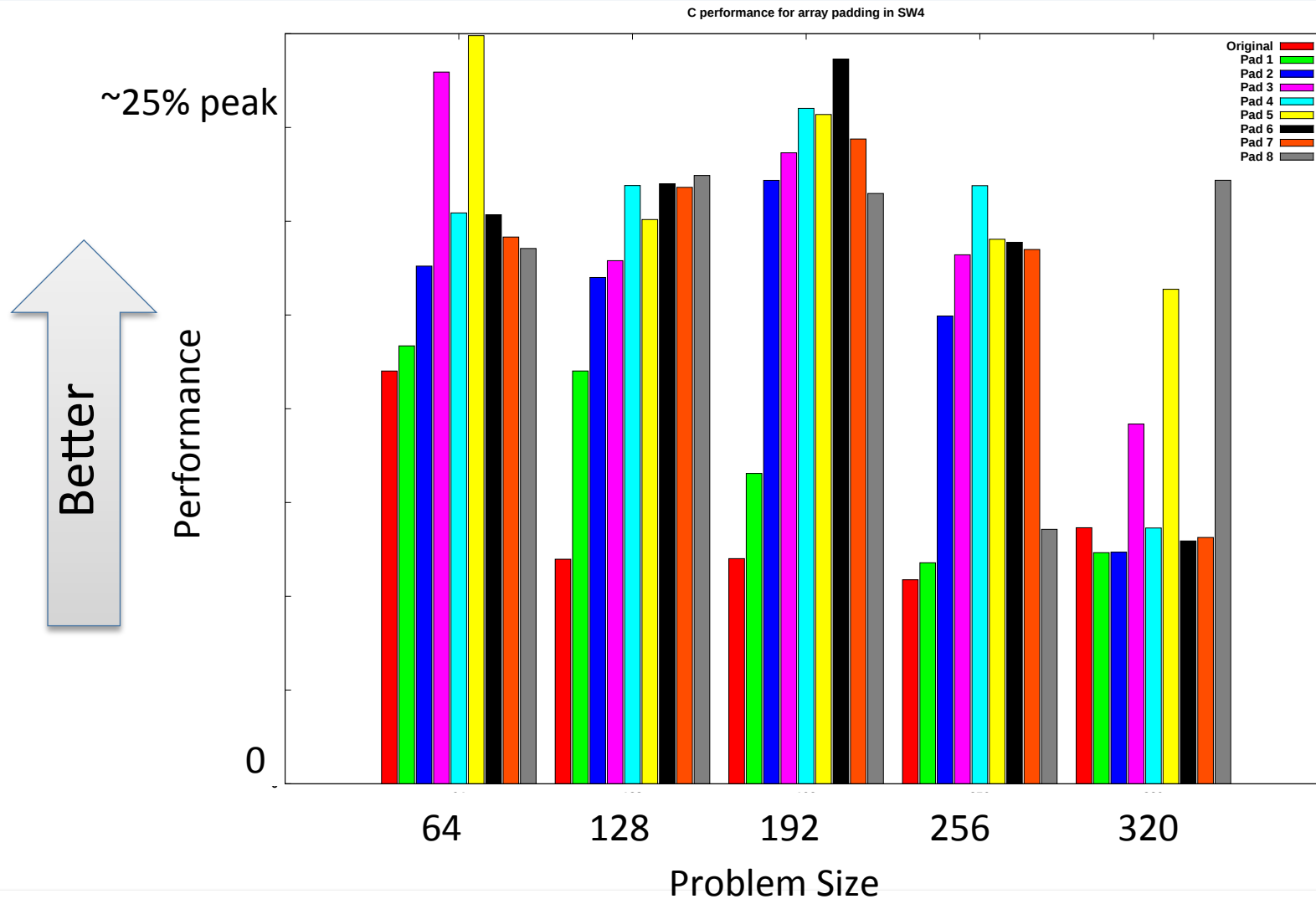


Over 200
lines of
similar code

Sw4 Haswell performance with index reordering



SW4 performance with various padding added



SW4 CPU Performance Observations

- The fortran vs C fight is fundamentally uninteresting to me
- Intel compiler does an impressive job with naïve code
- Original fortran version out performs C
 - C compiler misses a vectorization opportunity found by fortran compiler
 - If you don't want to think, use fortran
 - Or, complain to your C compiler vendor
- Fortan and C performance are equal with trivial index swap
 - May be better or worse than original fortran depending on problem size
 - Performance dependence on problem size can be fixed with padding
 - You can get better performance if you think

Stencil Optimization for GPUS

- Recent research on stencil optimization has focused on “time-tiling” and kernel fusion
 - Reduce data movement to/from main memory for bandwidth limited codes
- Higher-order stencils have higher arithmetic intensity
 - Conventional wisdoms says this should be easier to optimize
 - In practice, data re-use must be balanced with register pressure
- Initial generated code for SW4 exhibited very poor performance
 - HW counter analysis revealed massive register spilling
 - Suggested use of opposite approach to typical kernel fusion optimization for stencils: **kernel fission**
 - Kernel fission reduces the register pressure per kernel
 - But reduces data reuse and increases overall data traffic

Kernel Fission and Fusion

Fission:

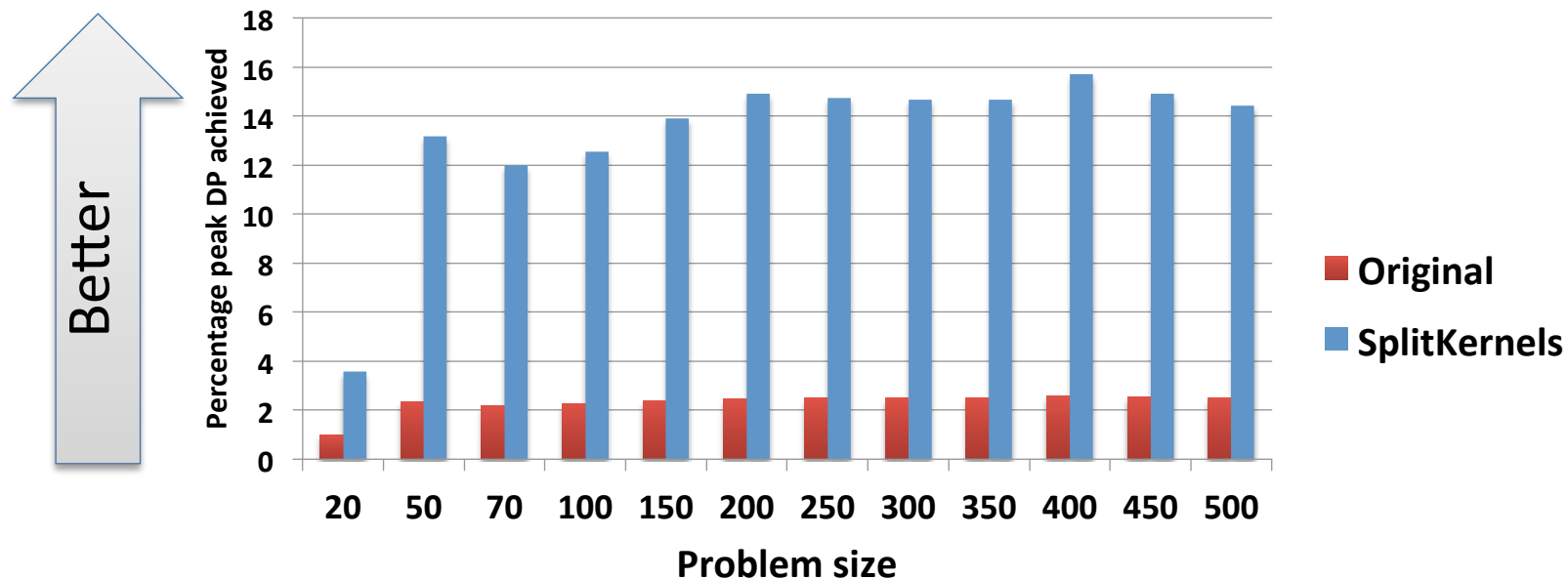
- Put each statement in a separate kernel
- Rewrite stencil statements as accumulations (statement splitting)
 - A sub-statement reads from minimal number of inputs
 $C = \text{stencil}(A, B); \longrightarrow C = \text{stencil}(A); C += \text{stencil}(B);$
 - If possible, split a sub-statement into stencils along only one dimension
 $C = \text{stencil}(A_{ij}); \longrightarrow C = \text{stencil}(A_i); C += \text{stencil}(A_j);$

Fusion:

- Regroup accumulations across kernels to increase reuse and shorten live ranges
- Group statements with stencil along the same dimension together
 - ♦ Statements with high reuse are lexicographically closer
- Fuse kernels with no reuse to minimize data movement without increasing register pressure
 - ♦ Group stencils along orthogonal dimensions together (no/low reuse implies low probability of register spills)

GPU Performance of SW4 Kernel

SW4



- Preliminary results for kernel from double precision (DP) rhs4th3fort routine of SW4
- Compiled on Tesla K40c with 64 registers per thread
- Spills for Original kernel: 1416 bytes spill stores, 1712 bytes spill loads
- Split-Kernel contains 3 kernels, each using 64 registers per thread: 0 Register spills

On Further Performance Improvement

- Kernel fission solves the register spill problem but...
 - Performance is only about 15% of roofline bound
 - Global memory access is not the bottleneck
 - Limiting factor seems latency/bandwidth of L1/L2 cache
- Challenges for SW4 stencils on GPUs
 - SW4 stencils are much more complex than single-array high-order stencils optimized previously
 - And this is the simplest stencil in SW4
 - Smaller per-thread capacity of L1/L2 caches on GPU compared to CPU
 - Cache performance worse on GPU compared to CPU
- Work is ongoing

Summary

- Code generation is a big part of many portability solutions
 - Raja, Kokkos, OpenMP, etc.
 - Best techniques and practices are still shaking out
 - Unreasonable to put the whole demand on vendor compiler
- Not all higher-order stencils are created equal
 - Need more cooperation between application developers and tool chain developers/researchers
- More registers would probably help

